**Storage Classes**

Storage Classes are used to describe the features of a variable/function. These features basically include the scope, visibility and life-time which help us to trace the existence of a particular variable during the runtime of a program. C language uses 4 storage classes, namely:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Storage Classes** | **Storage Place** | **Default Value** | **Scope** | **Lifetime** | **Declaration** |
| auto | RAM | Garbage Value | Local | Within function | Inside a function/block |
| extern | RAM | Zero | Global | Till the end of the main program Maybe declared anywhere in the program | Outside all functions |
| static | RAM | Zero | Local | Till the end of the main program, Retains value between multiple functions call | Inside a function/block |
| register | Register | Garbage Value | Local | Within the function | Inside a function/block |

**Automatic**

* Automatic variables are allocated memory automatically at runtime.
* The visibility of the automatic variables is limited to the block in which they are defined.
* The scope of the automatic variables is limited to the block in which they are defined.
* The automatic variables are initialized to garbage by default.
* The memory assigned to automatic variables gets freed upon exiting from the block.
* The keyword used for defining automatic variables is auto.
* Every local variable is automatic in C by default.

**Example 1**

#include <stdio.h>

int main()

{

int a; //auto

float c;

printf("%d %f",a,c); // printing initial default value of automatic variables a, b, and c.

return 0;

}

**Output:**

garbage garbage

**Example 2**

#include <stdio.h>

int main()

{

int a = 10, i;

printf("%d ",++a);

{

int a = 20;

for (i=0;i<3;i++)

{

printf("%d ",a); // 20 will be printed 3 times since it is the local value of a

}

}

printf("%d ",a); // 11 will be printed since the scope of a = 20 is ended.

}

**Output:**

11 20 20 20 11

**Static**

* The variables defined as static specifier can hold their value between the multiple function calls.
* Static local variables are visible only to the function or the block in which they are defined.
* A same static variable can be declared many times but can be assigned at only one time.
* Default initial value of the static integral variable is 0 otherwise null.
* The visibility of the static global variable is limited to the file in which it has declared.
* The keyword used to define static variable is static.
* They can also be used as a global variable

**Example 1**

#include<stdio.h>

static char c;

static int i;

static float f;

static char s[100];

void main ()

{

printf("%d %d %f",c,i,f); // the initial default value of c, i, and f will be printed.

}

**Output:**

0 0 0.000000

**Example 2**

#include<stdio.h>

void sum()

{

static int a = 10;

static int b = 24;

printf("%d %d \n",a,b);

a++;

b++;

}

void main()

{

int i;

for(i = 0; i< 3; i++)

{

sum(); // The static variables holds their value between multiple function calls.

}

}

**Output:**

10 24

11 25

12 26

**Register**

* The variables defined as the register is allocated the memory into the CPU registers depending upon the size of the memory remaining in the CPU.
* We cannot dereference the register variables, i.e., we cannot use &operator for the register variable.
* The access time of the register variables is faster than the automatic variables.
* The register keyword is used for the variable which should be stored in the CPU register. However, it is compiler’s choice whether or not; the variables can be stored in the register.
* We can store pointers into the register, i.e., a register can store the address of a variable.
* Static variables cannot be stored into the register since we cannot use more than one storage specifier for the same variable.
* You can use the register storage class when you want to store local variables within functions or blocks in CPU registers instead of RAM to have quick access to these variables.

**Example 1**

#include <stdio.h>

int main()

{

register int a; // variable a is allocated memory in the CPU register. The initial default value of a is garbage.

printf("%d",a);

}

**Output:**

garbage

**Example 2**

#include <stdio.h>

int main()

{

register int a = 0;

printf("%u",&a); // This will give a compile time error since we cannot access the address of a register variable.

}

**Output:**

main.c:5:5: error: address of register variable a requested

printf("%u",&a);

^~~~~~

**External**

* The external storage class is used to tell the compiler that the variable defined as extern is declared with an external linkage elsewhere in the program.
* The variables declared as extern are not allocated any memory. It is only declaration and intended to specify that the variable is declared elsewhere in the program.
* The default initial value of external integral type is 0 otherwise null.
* We can only initialize the extern variable globally, i.e., we cannot initialize the external variable within any block or method.
* An external variable can be declared many times but can be initialized at only once.
* If a variable is declared as external, then the compiler searches for that variable to be initialized somewhere in the program which may be extern or static. If it is not, then the compiler will show an error.
* Extern storage class is used when we have global functions or variables which are shared between two or more files.

**Example 1**

#include <stdio.h>

int main()

{

extern int a;

printf("%d",a);

}

**Output**

main.c:(.text+0x6): undefined reference to `a'

collect2: error: ld returned 1 exit status

**Example 2**

#include <stdio.h>

int a;

int main()

{

extern int a; // variable a is defined globally, the memory will not be allocated to a

printf("%d",a);

}

**Output**

0

**Example 3**

#include <stdio.h>

int a;

int main()

{

extern int a = 0; // this will show a compiler error since we cannot use extern and initializer at same time

printf("%d",a);

}

**Output**

compile time error

main.c: In function ?main?:

main.c:5:16: error: ?a? has both ?extern? and initializer

extern int a = 0;

**Example 4**

#include <stdio.h>

int main()

{

extern int a; // Compiler will search here for a variable a defined and initialized somewhere in the pogram or not.

printf("%d",a);

}

int a = 20;

**Output**

20

**Type Casting**

Type Casting is basically a process in C in which we change a variable belonging to one data type to another one. Converting one datatype into another is known as type casting or, type-conversion. For example, if you want to store a 'long' value into a simple integer then you can type cast 'long' to 'int'. You can convert the values from one type to another explicitly using the cast operator as follows −

(type\_name) expression

Consider the following example where the cast operator causes the division of one integer variable by another to be performed as a floating-point operation −

#include <stdio.h>

main() {

int sum = 17, count = 5;

double mean;

mean = (double) sum / count;

printf("Value of mean : %f\n", mean );

}

Value of mean: 3.400000

It should be noted here that the cast operator has precedence over division, so the value of sum is first converted to type double and finally it gets divided by count yielding a double value.

Type conversions can be implicit which is performed by the compiler automatically, or it can be specified explicitly through the use of the cast operator. It is considered good programming practice to use the cast operator whenever type conversions are necessary.

**Example**

Without Type Casting:

int f= 9/4;

printf("f : %d\n", f );//Output: 2

With Type Casting:

float f=(float) 9/4;

printf("f : %f\n", f );//Output: 2.250000

The process of type casting can be performed in two major types in a C program. These are:

1. **Implicit**

Implementing the implicit type casting is very easy in a program. We use it to convert the data type of any variable without losing the actual meaning that it holds. The implicit type casting occurs automatically. In simpler words, the implicit type casting performs the conversion without altering any of the values stored in the program’s variables.

Follow these points to understand what rules the implicit type casting follows in a C program:

* If we are performing a conversion on two of the different data types in a program, then the conversion of the lower data type to the higher data type will occur automatically.
* If we suppose that we are performing the type casting operation among two different data types like float and int, then the resultant value would be the floating data type (float).

int x = 4;

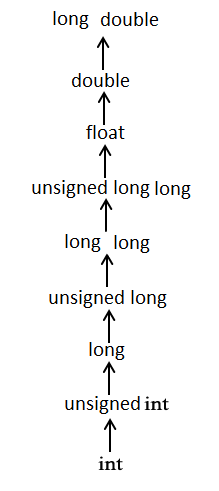
float a = 12.4, b;

b = a / x;

In the example given above, the variable x will get converted to the float data type (float) automatically, and it is comparatively a bigger data type in C. Here, the variable x and the variable a would be equal in terms of their data types. Thus, the value of b would be equal to 12.4/4.0=3.1.

**Usual Arithmetic Conversion**

The usual arithmetic conversions are implicitly performed to cast their values to a common type. The compiler first performs integer promotion; if the operands still have different types, then they are converted to the type that appears highest in the following hierarchy −



The usual arithmetic conversions are not performed for the assignment operators, nor for the logical operators && and ||. Let us take the following example to understand the concept −

#include <stdio.h>

main() {

int i = 17;

char c = 'c'; /\* ascii value is 99 \*/

float sum;

sum = i + c;

printf("Value of sum : %f\n", sum );

}

When the above code is compiled and executed, it produces the following result −

Value of sum : 116.000000

Here, it is simple to understand that first c gets converted to integer, but as the final value is double, usual arithmetic conversion applies and the compiler converts i and c into 'float' and adds them yielding a 'float' result.

1. **Explicit**

This process is not at all like the implicit type casting in C, where the conversion of data type occurs automatically. Conversely, in the case of explicit type casting, the programmer needs to force the conversion. In simpler words, one has to perform type casting on the data types on their own.

Syntax:

(name\_of\_data\_type) expression

Here, name\_of\_data\_type refers to the name of that data type to which we want to convert the available data type in the code. This expression can be a constant, a variable, or even an actual expression in a program.

float num = 56.3;

int p = (int)num + 50; // data type casting explicitly

printf(“The value of the digit used is: %f\n”, num);

printf(“The value of the variable p is: %d\n”,p);

**Output:**

The value of the digit used is: 56.299999

The value of the variable p is: 106

**Integer Promotion**

Integer promotion is the process by which values of integer type "smaller" than int or unsigned int are converted either to int or unsigned int. Consider an example of adding a character with an integer −

#include <stdio.h>

main() {

int i = 17;

char c = 'c'; /\* ascii value is 99 \*/

int sum;

sum = i + c;

printf("Value of sum : %d\n", sum );

}

When the above code is compiled and executed, it produces the following result −

Value of sum : 116

Here, the value of sum is 116 because the compiler is doing integer promotion and converting the value of 'c' to ASCII before performing the actual addition operation.

**Difference Between Type Casting and Type Conversion**

|  |  |  |
| --- | --- | --- |
| **S.N.** | **Type Casting** | **Type Conversion** |
| 1 | Type casting is a mechanism in which one data type is converted to another data type using a casting () operator by a programmer. | Type conversion allows a compiler to convert one data type to another data type at the compile time of a program or code. |
| 2 | It can be used both compatible data type and incompatible data type. | Type conversion is only used with compatible data types, and hence it does not require any casting operator. |
| 3 | It requires a programmer to manually casting one data into another type. | It does not require any programmer intervention to convert one data type to another because the compiler automatically compiles it at the run time of a program. |
| 4 | It is used while designing a program by the programmer. | It is used or take place at the compile time of a program. |
| 5 | When casting one data type to another, the destination data type must be smaller than the source data. | When converting one data type to another, the destination type should be greater than the source data type. |
| 6 | It is also known as narrowing conversion because one larger data type converts to a smaller data type. | It is also known as widening conversion because one smaller data type converts to a larger data type. |
| 7 | It is more reliable and efficient. | It is less efficient and less reliable. |
| 8 | There is a possibility of data or information being lost in type casting. | In type conversion, data is unlikely to be lost when converting from a small to a large data type. |
| 8 | float b = 3.0;  int a = (int) b | int x = 5, y = 2, c;  float q = 12.5, p;  p = q/x; |

**Volatile**

A volatile keyword in C is nothing but a qualifier that is used by the programmer when they declare a variable in source code. It is used to inform the compiler that the variable value can be changed any time without any task given by the source code. Volatile is usually applied to a variable when we are declaring it. The main reason behind using volatile keyword is that it is used to prevent optimizations on objects in our source code. Therefore, an object declared as volatile can’t be optimized because its value can be easily changed by the code. As a general rule, all variables whose value can change suddenly due to an external reason (like an external device or another program thread) should be declared using volatile. This is mainly used in Embedded Systems programming and real time systems.

**Syntax**

volatile data\_type variable\_name ;

volatile data\_type \*variable\_name ;

**Explanation:**In the above declaration volatile keyword is mandatory to be used then data\_type means any data type it can be wither integer, float, or double. Finally, the name of the variable as per our choice.

volatile int x ;

volatile int \*a;

**Example #1**

**Without using keyword Volatile:**

#include<stdio.h> // C header file for standard input and output

int a = 0 ; // initilaizing and declaring the integer a to value 0.

int main () // main class

{

if ( a == 0 ) // This condition will be true

{

printf ( " a = 0 \n " ) ;

}

else // Else part will be optimized

{

printf ( " a ! = 0 \n " ) ;

}

return 0 ; // returning value

}

**Output:**

a=0

**Explanation:** In the above code, we have declared an integer variable with value 0 assigned to it. Then in the main class, we have set the if condition which will hold true until and unless the value of variable a is 0. As you can see the output will always be 0 as the condition will always remain true so that that code won’t move to the else part as it will ignore the else part.

**Example #2**

**With using keyword Volatile:**

#include<stdio.h>

volatile int a ; /\* volatile Keyword used before declaration of integer variable a \*/

int main() // main class

{

a = 0 ; // initializing the integer value to 0

if (a == 0) // applying if condition

{

printf ( " a = 0 \n " ) ;

}

else// Now compiler never optimize else part because the variable is declared as volatile

{

printf ( " a ! = 0 \n " ) ;

}

return 0 ;

}

**Output:**

a=0

**Explanation:** In the above code, we have declared a volatile integer variable a. Then in the main class, we have set two things one is the value of integer variable is 0 and second is the if condition which will hold true until and unless the value of variable a is 0. As you can see the output will always be 0 as the condition will always remain true because the variable is declared as volatile. Therefore, the compiler won’t optimize the else part of code because of the volatile keyword used before integer. So the compiler will know that the variable can change anytime. Hence, it will read the else part as the final executable code and display the result.

**Enumeration**

The enum in C is also known as the enumerated type. It is a user-defined data type that consists of integer values, and it provides meaningful names to these values. The use of enum in C makes the program easy to understand and maintain. The enum is defined by using the enum keyword.

The following is the way to define the enum in C:

enum flag{integer\_const1, integer\_const2,.....integter\_constN};

In the above declaration, we define the enum named as flag containing 'N' integer constants. The default value of integer\_const1 is 0, integer\_const2 is 1, and so on. We can also change the default value of the integer constants at the time of the declaration.

**Example**

enum fruits{mango, apple, strawberry, papaya};

The default value of mango is 0, apple is 1, strawberry is 2, and papaya is 3. If we want to change these default values, then we can do as given below:

enum fruits{

mango=2,

apple=1,

strawberry=5,

papaya=7,

};

**Enumerated Type Declaration**

We can declare the variable of a user-defined data type, such as enum. Let's see how we can declare the variable of an enum type.

Suppose we create the enum of type status as shown below:

enum status{false,true};

Now, we create the variable of status type:

enum status s; // creating a variable of the status type.

To create a variable, the above two statements can be written as:

enum status

{

false,true

} s;

In this case, the default value of false will be equal to 0, and the value of true will be equal to 1.

**Example**

#include <stdio.h>

enum weekdays{Sunday=1, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday};

int main()

{

enum weekdays w; // variable declaration of weekdays type

w=Monday; // assigning value of Monday to w.

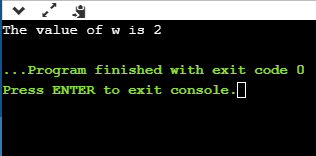
printf("The value of w is %d",w);

return 0;

}

In the above code, we create an enum type named as weekdays, and it contains the name of all the seven days. We have assigned 1 value to the Sunday, and all other names will be given a value as the previous value plus one.

**Output**



**Why do we use enum?**

The enum is used when we want our variable to have only a set of values. For example, we create a direction variable. As we know that four directions exist (North, South, East, West), so this direction variable will have four possible values. But the variable can hold only one value at a time. If we try to provide some different value to this variable, then it will throw the compilation error.

The enum is also used in a switch case statement in which we pass the enum variable in a switch parenthesis. It ensures that the value of the case block should be defined in an enum.

#include <stdio.h>

enum days{sunday=1, monday, tuesday, wednesday, thursday, friday, saturday};

int main()

{

enum days d;

d=monday;

switch(d)

{

case sunday:

printf("Today is sunday");

break;

case monday:

printf("Today is monday");

break;

case tuesday:

printf("Today is tuesday");

break;

case wednesday:

printf("Today is wednesday");

break;

case thursday:

printf("Today is thursday");

break;

case friday:

printf("Today is friday");

break;

case saturday:

printf("Today is saturday");

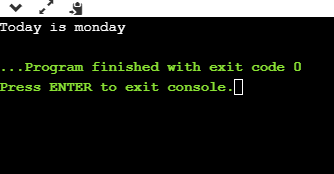
break;

}

return 0;

}

**Output**



**Some important points related to enum**

* The enum names available in an enum type can have the same value. Let's look at the example.

#include <stdio.h>

int main()

{

enum fruits{mango = 1, strawberry=0, apple=1};

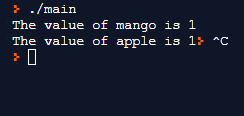
printf("The value of mango is %d", mango);

printf("\nThe value of apple is %d", apple);

return 0;

}

**Output**

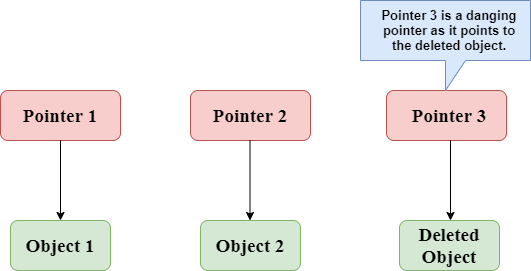


* If we do not provide any value to the enum names, then the compiler will automatically assign the default values to the enum names starting from 0.
* We can also provide the values to the enum name in any order, and the unassigned names will get the default value as the previous one plus one.
* The values assigned to the enum names must be integral constant, i.e., it should not be of other types such string, float, etc.

**Dangling Pointers in C**

The most common bugs related to pointers and memory management is dangling/wild pointers. Sometimes the programmer fails to initialize the pointer with a valid address, then this type of initialized pointer is known as a dangling pointer in C.

Dangling pointer occurs at the time of the object destruction when the object is deleted or de-allocated from memory without modifying the value of the pointer. In this case, the pointer is pointing to the memory, which is de-allocated. The dangling pointer can point to the memory, which contains either the program code or the code of the operating system. If we assign the value to this pointer, then it overwrites the value of the program code or operating system instructions; in such cases, the program will show the undesirable result or may even crash. If the memory is re-allocated to some other process, then we dereference the dangling pointer will cause the segmentation faults.



In the above figure, we can observe that the Pointer 3 is a dangling pointer. Pointer 1 and Pointer 2 are the pointers that point to the allocated objects, i.e., Object 1 and Object 2, respectively. Pointer 3 is a dangling pointer as it points to the de-allocated object.

**Using free() function to de-allocate the memory.**

#include <stdio.h>

int main()

{

int \*ptr=(int \*)malloc(sizeof(int));

int a=560;

ptr=&a;

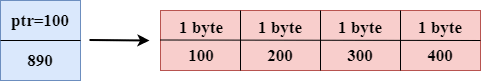
free(ptr);

return 0;

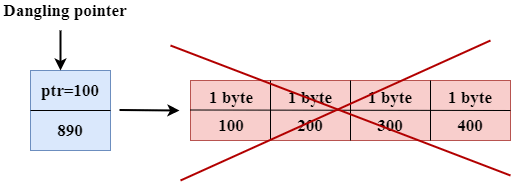
}

In the above code, we have created two variables, i.e., \*ptr and a where 'ptr' is a pointer and 'a' is a integer variable. The \*ptr is a pointer variable which is created with the help of malloc() function. As we know that malloc() function returns void, so we use int \* to convert void pointer into int pointer.

The statement int \*ptr=(int \*)malloc(sizeof(int)); will allocate the memory with 4 bytes shown in the below image:



The statement free(ptr) de-allocates the memory as shown in the below image with a cross sign, and 'ptr' pointer becomes dangling as it is pointing to the de-allocated memory.



If we assign the NULL value to the 'ptr', then 'ptr' will not point to the deleted memory. Therefore, we can say that ptr is not a dangling pointer, as shown in the below image:



**Variable goes out of the scope**

When the variable goes out of the scope then the pointer pointing to the variable becomes a dangling pointer.

#include<stdio.h>

int main()

{

char \*str;

{

char a = ‘A’;

str = &a;

}

// a falls out of scope

// str is now a dangling pointer

printf("%c", \*str);

}

**Output:** A

In the above code, we did the following steps:

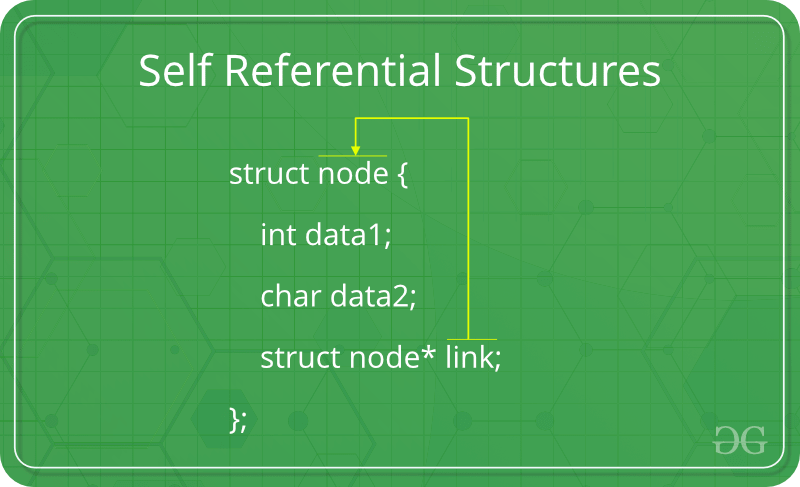
* First, we declare the pointer variable named 'str'.
* In the inner scope, we declare a character variable. The str pointer contains the address of the variable 'a'.
* When the control comes out of the inner scope, 'a' variable will no longer be available, so str points to the de-allocated memory. It means that the str pointer becomes the dangling pointer.

**Avoiding Dangling Pointer Errors**

The dangling pointer errors can be avoided by initializing the pointer to the NULL value. If we assign the NULL value to the pointer, then the pointer will not point to the de-allocated memory. Assigning NULL value to the pointer means that the pointer is not pointing to any memory location.

**Self-Referential Structures**

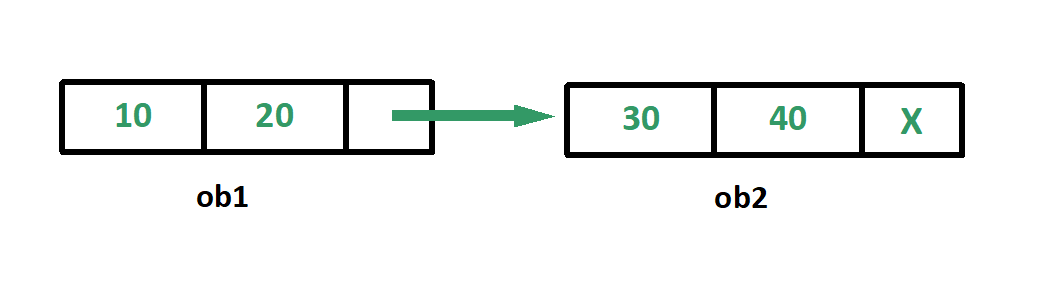
Self-Referential structures are those structures that have one or more pointers which point to the same type of structure, as their member.



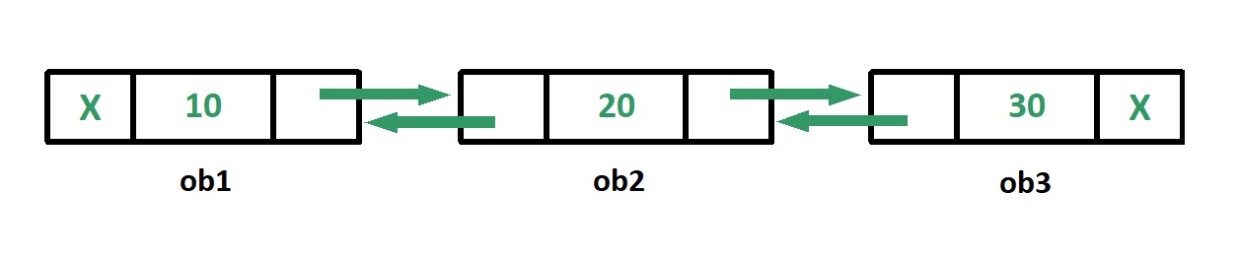
In other words, structures pointing to the same type of structures are self-referential in nature.

**Types of Self Referential Structures**

**Self-Referential Structure with Single Link:** These structures can have only one self-pointer as their member. The following example will show us how to connect the objects of a self-referential structure with the single link and access the corresponding data members. The connection formed is shown in the following figure.



**Self-Referential Structure with Multiple Links:** Self-referential structures with multiple links can have more than one self-pointers. Many complicated data structures can be easily constructed using these structures. Such structures can easily connect to more than one nodes at a time. The following example shows one such structure with more than one links.



**Applications:** Self-referential structures are very useful in creation of other complex data structures like: Linked Lists, Stacks, Queues, Trees, Graphs etc. Unlike a static data structure such as array where the number of elements that can be inserted in the array is limited by the size of the array, a self-referential structure can dynamically be expanded or contracted.

**Dynamic Memory Allocation in C**

The concept of dynamic memory allocation in c language enables the C programmer to allocate memory at runtime. Dynamic memory allocation in C language is possible by 4 functions of stdlib.h header file.

* malloc()
* calloc()
* realloc()
* free()

|  |  |
| --- | --- |
| **Static Memory Allocation** | **Dynamic Memory Allocation** |
| memory is allocated at compile time. | memory is allocated at run time. |
| memory can't be increased while executing program. | memory can be increased while executing program. |
| used in array. | used in linked list. |

Now let's have a quick look at the methods used for dynamic memory allocation.

|  |  |
| --- | --- |
| malloc() | allocates single block of requested memory. |
| calloc() | allocates multiple block of requested memory. |
| realloc() | reallocates the memory occupied by malloc() or calloc() functions. |
| free() | frees the dynamically allocated memory. |

**malloc() function in C-** The malloc() function allocates single block of requested memory.

It doesn't initialize memory at execution time, so it has garbage value initially. It returns NULL if memory is not sufficient.

The syntax of malloc() function is given below:

ptr=(cast-type\*)malloc(byte-size)

**calloc() function in C-** The calloc() function allocates multiple block of requested memory.

It initially initialize all bytes to zero. It returns NULL if memory is not sufficient.

The syntax of calloc() function is given below:

ptr=(cast-type\*)calloc(number, byte-size)

**realloc() function in C-** If memory is not sufficient for malloc() or calloc(), you can reallocate the memory by realloc() function. In short, it changes the memory size.

ptr=realloc(ptr, new-size)

**free() function in C-** The memory occupied by malloc() or calloc() functions must be released by calling free() function. Otherwise, it will consume memory until program exit.

free(ptr)

**C Pre-processor Directives**

The C pre-processor is a microprocessor that is used by compiler to transform your code before compilation. It is called micro pre-processor because it allows us to add macros. All pre-processor directives starts with hash # symbol.

**Note: Pre-processor directives are executed before compilation**.

Let's see a list of pre-processor directives.

* #include
* #define
* #undef
* #ifdef
* #ifndef
* #if
* #else
* #elif
* #endif
* #error
* #pragma

**C #include**

The #include pre-processor directive is used to paste code of given file into current file. It is used include system-defined and user-defined header files. If included file is not found, compiler renders error.

By the use of #include directive, we provide information to the pre-processor where to look for the header files. There are two variants to use #include directive.

* #include <filename>
* #include "filename"

The #include <filename> tells the compiler to look for the directory where system header files are held.

The #include "filename" tells the compiler to look in the current directory from where program is running.

**C #define**

The #define pre-processor directive is used to define constant or micro substitution. It can use any basic data type. Do NOT put a semicolon character at the end of #define statements.

Syntax:

#define token value

#include <stdio.h>

#define PI 3.14

main() {

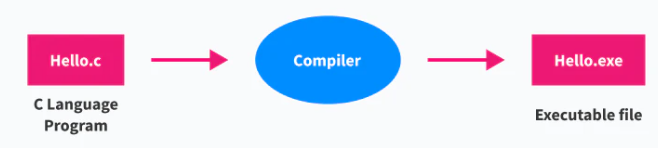
printf("%f",PI);

}

**Output:**

3.140000

To understand how and why pre-processor directives are executed before compilation, let us look at the process of how the whole compilation process works in a C Program. Let's suppose we have written a hello.c program to print Hello, World! in the output. The compilation process will generate an executable file, hello.exe from our hello.c program file.

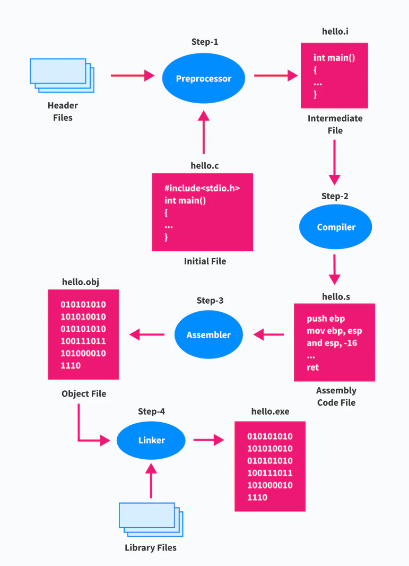


**Compilation Process**

It is a process of converting Human Understandable (High Level) Code into Machine Understandable (Low Level) Code. Let us look at the steps involved in the compilation process.

* Step 1, We have a written C Program file with an extension of .c i.e. hello.c file.
* Step 2 is pre-processing of header files, all the statements starting with # (hash symbol) are replaced during the compilation process with the help of a pre-processor. It generates an intermediate file with **.i file** extension i.e. a hello.i file.
* Step 3 is a compilation of hello.i file, compiler software translates the hello.i file to hello.s file having assembly-level instructions (low-level code).
* Step 4, assembly-level code instructions are converted into a machine-understandable code (binary/hexadecimal form) by the assembler, and the file generated is known as the object file with an extension of .obj i.e. hello.obj file.
* Step 5, Linker is used to link the library files with the object file to define the unknown statements. It generates an executable file with .exe extension i.e. a hello.exe file.
* Next, we can run the hello.exe executable file to get the desired output on our output window.

The below diagram shows all the steps involved in the compilation process.



**C Macros**

A macro is a segment of code which is replaced by the value of macro. Macro is defined by #define directive. There are two types of macros:

* Object-like Macros
* Function-like Macros

**Object-like Macros-** The object-like macro is an identifier that is replaced by value. It is widely used to represent numeric constants. For example:

#define PI 3.14

Here, PI is the macro name which will be replaced by the value 3.14.

**Function-like Macros-**The function-like macro looks like function call. For example:

#define MIN(a,b) ((a)<(b)?(a):(b))

Here, MIN is the macro name.

**C Predefined Macros**

ANSI C defines many predefined macros that can be used in c program.

|  |  |  |
| --- | --- | --- |
| **No.** | **Macro** | **Description** |
| 1 | \_DATE\_ | represents current date in "MMM DD YYYY" format. |
| 2 | \_TIME\_ | represents current time in "HH:MM:SS" format. |
| 3 | \_FILE\_ | represents current file name. |
| 4 | \_LINE\_ | represents current line number. |
| 5 | \_STDC\_ | It is defined as 1 when compiler complies with the ANSI standard. |

#include<stdio.h>

int main(){

printf("File :%s\n", \_\_FILE\_\_ );

printf("Date :%s\n", \_\_DATE\_\_ );

printf("Time :%s\n", \_\_TIME\_\_ );

printf("Line :%d\n", \_\_LINE\_\_ );

printf("STDC :%d\n", \_\_STDC\_\_ );

return 0;

}

**Output:**

File :simple.c

Date :Dec 6 2015

Time :12:28:46

Line :6

STDC :1